Dear CS124 Grader,

About the Dataset and City Class

This dataset is a collection of the largest cities in the US by population. Each city by default is listed in order from the greatest population to the least, however for this project the cities were sorted in alphabetical order so that the population would not be in order from the first city to the last. This data was collected in 2013, and is one of the more recent data sets available already in the form of a .txt or a .csv file. This data is also very interesting. Cities are constantly growing and shrinking at the same time, especially these large cities. Each entry has 5 attributes, including the city name (string), the state of the city (string), the population of the city (integer), the rank of the city in terms of population (integer), and the percent growth in terms of population from 2000 to 2013 (double). Each of these attributes are fields of my City class.

There are getters and setters because there is no sensitive data. There is no input checking, because all data is perfectly sorted into a combination of strings, integers, and doubles.

The << operator has been overloaded, so each city can be listed in a consistent way in the console. The >, <, and == operators have also been overloaded so a City can be compared to another City.

Search Efficiency in Different Trees

This dataset was used in a series of tests to determine which tree structure would be best for certain scenarios of use. Below is a summary of the data collected when searches were performed. For each tree, searches were performed for various elements in the tree in 3 different orders; in a sorted order, a random order, and a sorted order where each sort was repeated 5 times in a row. The storage depths of nodes from each search were recorded to separate text files, and are summarized and graphed below.

There were also 3 “dummy objects” that were searched for in each tree. These dummy objects were never actually placed in any of the trees, so this was essentially a test of the average depth that needed to be reached before it became apparent that the node didn’t exist in the tree.

Binary Search Tree

Code Source:

Weiss, Mark Allen. “Source Code for Data Structures and Algorithm Analysis in C++ (Fourth Edition).” CIU, 4th Edition, Ciu.edu, users.cs.fiu.edu/~weiss/dsaa_c++4/code/.

As stated above, the code for the tree is from fiu.edu, but was modified to work for this specific project. The contains method was modified to take a depth parameter, and increment as the search for a node went deeper into the tree. This exact method of tracking depth was also used for the AVL Tree. Below are the depths and depth graph representations of nodes that were searched for in each of the three search orders.

Complexity

As discussed in class, if a binary search tree has a height of h, searching through the tree has O(h) worst-case runtime complexity. A binary search tree with n nodes has a minimum of O(log n) levels, so it takes at least O(log n) searches to find a particular node.

Searching for Elements in Sorted and Random Orders

The binary search tree seems to be the least efficient of the three trees. At least, in terms of depth. When searching for vector elements in order, the average storage/search depth for the binary search tree is 11.15852443. The same held true for searching in a random order. These were the highest averages, which means more traversing of the tree was necessary to reach the destination nodes than both other tree types.

Searching for Elements in Succession

As for searching for the same element multiple (5) times in a row, the average search depth was lower, but only because only the first 20% of searches were carried out. The average was 8.065802592, which was the highest average compared to other trees.

Dummy Objects

Searching for the “dummy objects” was also the least efficient. For the searches of the dummy objects, depths of 15 were all returned, meaning searches of “15 levels down” were conducted before it was determined that the objects were not in the tree. This is more than both the AVL tree and the splay tree.

AVL Tree

Code Source:

Weiss, Mark Allen. “Source Code for Data Structures and Algorithm Analysis in C++ (Fourth Edition).” CIU, 4th Edition, Ciu.edu, users.cs.fiu.edu/~weiss/dsaa_c++4/code/.

Just like with the binary search tree, the code for the tree is from fiu.edu, but was modified to work for this specific project. The contains method was modified to take a depth parameter, and increment as the search for a node went deeper into the tree. This exact method of tracking depth was also used for the binary search tree Tree. Below are the depths and depth graph representations of nodes that were searched for in each of the three search orders.

Complexity

An AVL tree is just a binary search tree, with subtrees differering in height by no more than 1. Since the height of an AVL tree of height n is O(log n), insertion worst case would be O(log n). Searching through an AVL tree is identical to a normal binary search tree.

Searching for Elements in Sorted and Random Orders

The AVL Tree seems to be more efficient than the binary search tree, but less efficient than the splay tree. At least, in terms of depth. When searching for vector elements in order, the average storage/search depth for the AVL tree is 8.180458624. The same holds true for random searching. These were the lowest averages for searching in these orders, which means more traversing of the tree was necessary to reach the destination nodes than the splay tree, and less than the binary search tree.

Searching for Elements in Succession

As for searching for the same element 5 multiple (5) times in a row, the average search depth was 6.897308076. Once again, this is only because only the first 20% of the vector was actually searched for. This is the second-lowest average for this category, only behind the splay tree.

Dummy Objects

Efficiency in searching for the "dummy objects" also came in second place, behind only the splay tree. For the searches of the dummy objects, depths of 11 were all returned, meaning searches of "11 levels down" were conducted before it was determined that the objects were not in the tree. This is less than the binary search tree, and more than the splay tree.

Splay Tree

Code Source:

Weiss, Mark Allen. "Source Code for Data Structures and Algorithm Analysis in C++ (Fourth Edition)." CIU, 4th Edition, Ciu.edu, users.cs.fiu.edu/~weiss/dsaa_c++4/code/.

As stated above, the code for the tree is from fiu.edu, but was modified to work for this specific project. The contains method was modified to take a depth parameter, pass it to the splay method. The splay method was modified so that depth was incremented every time the internal for loop iterated. Below are the depths and depth graph representations of nodes that were searched for in each of the three search orders.

Complexity

A splay tree is just like a binary search tree, except when a node is accessed, it is moved to the root. This means that m operations in splay tree takes on complexity time of O(m log n).

Searching for Elements in Sorted and Random Orders

The splay tree seems to be the most efficient than the binary search tree and the AVL tree. At least, in terms of depth. When searching for vector elements in order, the average storage/search depth for the splay tree is 8.754735793. When searching randomly, the average is 8.584247258. This difference is due to the splaying of the tree during every search. After a node is accessed, it is moved to the root for easier access in the future. Both of these were the second-lowest averages, which means less traversing of the tree was necessary to reach the destination node than the binary search tree, but more than the AVL tree.

Searching for Elements in Succession

When searching for the same element 5 multiple (5) times in a row, the splay tree was most efficient by far. This is once again due to the actual splaying of the tree, which moves the accessed node to the root. When one node is accessed 5 times in a row, that node is moved to the root the first access. For the next four searches, the node will already be at the root of the tree. This is by far the best tree to use when accessing the same node frequently.

Dummy Objects

The numbers were lower for the “dummy objects.” For the searches of the dummy objects, depths of 7, 2, and 1 were returned, meaning searches of 7, 2, and 1 “level(s) down” were conducted before it was determined that the objects were not in the tree. This is less than the binary search tree and the AVL tree.

Conclusions

Each of the trees returned much different results in terms of storage/search depth. When searching for elements in sorted or random orders, the best tree to use to retain small subtree sizes, is the AVL tree. When the same node in a tree has to be accessed multiple times at once, the splay tree is the most efficient tree to use.

Sincerely,
Max Peck